پیوست سوم

ساختمان‌داده‌ها برای مجموعه‌های از هم جدا

الگوریتم کروسکال (الگوریتم ۴-۲ بخش ۴-۱-۲) مستلزم این است که زیرمجموعه‌های از هم جدا ایجاد کنیم که هر کدام حاوی رأس متمایزی در گراف باشند و زیرمجموعه‌ها را مکرراً ادغام کنیم تا تمام رأس‌ها در یک مجموعه باشند. برای پیاده‌سازی این الگوریتم، به ساختمان‌داده‌ای برای مجموعه‌های از هم جدا نیاز داریم. کاربردهای دیگری از مجموعه‌های از هم جدا وجود دارد. به‌عنوان‌مثال، می‌توانند در بخش ۴-۳ برای بهبود پیچیدگی زمانی الگوریتم ۴-۴ (زمان‌بندی با مهلت معین) به کار روند.

به یاد داشته باشید که **نوع داده انتزاعی** شامل مجموعه‌ای از اشیا و عملیات مجاز بر روی آن‌ها است. قبل از پیاده‌سازی نوع داده انتزاعی مجموعه از هم جدا، باید اشیا و عملیات مورد نیاز را مشخص کنیم. با مجموعه جهانی U شروع می‌کنیم. به‌عنوان‌مثال، می‌توانیم داشته باشیم:

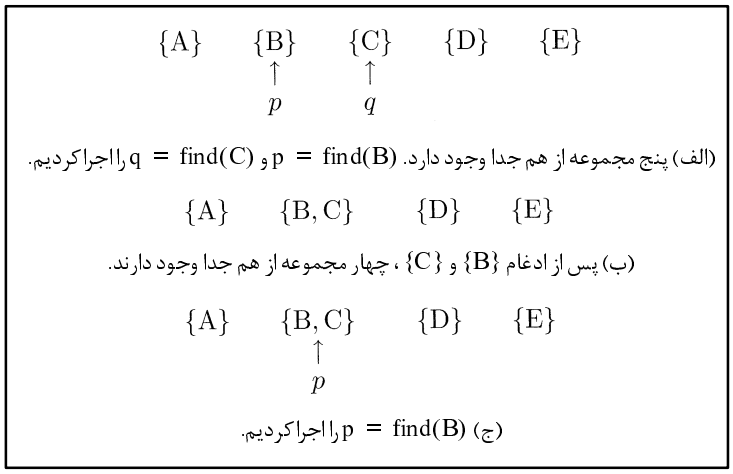
$$U = \{A, B, C, D, E\}$$

سپس می‌خواهیم رویه‌ای به‌نام makeset ایجاد کنیم که مجموعه‌ای از هر عنصر U تولید نماید. مجموعه‌های از هم جدای شکل ج ـ ۱ (الف) باید با فراخوانی‌های زیر ایجاد شوند:

```
for (each x ∈ U)
  makeset(x);
```

نیاز به نوع set_pointer و تابع find داریم، به‌طوری‌که اگر p و q از نوع set_pointer باشند، و فراخوانی‌های زیر را داشته باشیم:

```
p = find('B');
q = find('C');
```



شکل ج ـ ۱ نمونه‌ای از ساختمان‌داده مجموعه از هم جدا.

آن‌گاه p باید به مجموعه حاوی B، و q باید به مجموعه حاوی C اشاره کند. این موضوع در شکل ج ـ ۱(الف) آمده است. همچنین به رویه merge نیاز داریم تا دو مجموعه را در یک مجموعه ادغام کند. به عنوان مثال، فرض کنید اعمال زیر را انجام دهیم:

```
p = find('B');
q = find('C');
merge(p, q);
```

در این صورت، شکل ج ـ ۱(الف) باید به مجموعه‌های شکل ج ـ ۱(ج) تبدیل شوند. با توجه به مجموعه‌های از هم جدای شکل ج ـ ۱(ب)، فرض کنید فراخوانی زیر را انجام دهیم:

```
p = find('B');
```

نتیجه این فراخوانی در شکل ج ـ ۱(ج) آمده است. سرانجام، به رویه equal نیاز داریم تا بررسی کند که آیا دو مجموعه یکسان‌اند یا خیر. به‌عنوان‌مثال، فرض کنید مجموعه‌های شکل ج ـ ۱(ب) را داشته باشیم و فراخوانی‌های زیر را انجام دهیم:

```
p = find('B');
q = find('C');
r = find('A');
```

در این صورت، equal(p, q) باید مقدار true و equal(p, r) باید مقدار false را برگرداند.

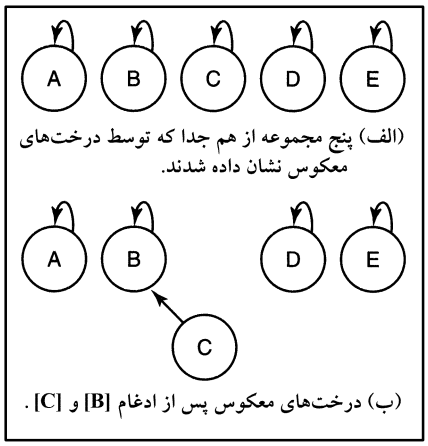
یک نوع داده انتزاعی را مشخص کردیم که اشیای آن شامل عناصر مجموعه جهانی و مجموعه‌های از هم جدای آن عناصر است و عملیات آن makset، find، merge و equal می‌باشند.

یک روش نمایش مجموعه‌های از هم جدا، استفاده از درخت‌هایی با اشاره گرهای معکوس است. در این درخت‌ها، هر گره غیر ریشه، به والد خود اشاره می‌کند. درحالی‌که هر ریشه به خودش اشاره می‌نماید. شکل ج ـ ۲(الف) درخت‌های متناظر با مجموعه‌های از هم جدای شکل ج ـ ۱(الف) را نشان می‌دهند و شکل ج ـ ۲(ب) درخت‌های متناظر با مجموعه‌های از هم جدای شکل ج ـ ۱(ب) را نمایش می‌دهند. برای پیاده‌سازی ساده‌ی این

درخت‌ها، فرض می‌کنیم مجموعه جهانی فقط حاوی اندیس‌ها (از نوع صحیح) است. برای بسط این پیاده‌سازی به مجموعه جهانی متناهی دیگر، فقط کافی است اندیسی به عناصر موجود در آن مجموعه جهانی داشته باشیم. درخت‌ها را می‌توان با استفاده از آرایه U نشان داد که در آن، هر اندیس به U، اندیسی در مجموعه جهانی است. اگر اندیس i گره غیر ریشه را نشان دهد، مقدار U[i] اندیس نشان‌دهنده والد آن است. اگر اندیس i نشان‌دهنده ریشه باشد، مقدار U[i] برابر با i است، به عنوان مثال، اگر ۱۰ اندیس در مجموعه جهانی باشد، آن‌ها را در آرایه‌ای از اندیس‌های U ذخیره می‌کنیم که از ۱ تا ۱۰ اندیس‌گذاری می‌شود.

برای قراردادن تمام اندیس‌ها در مجموعه‌های از هم جدا، برای هر i، داریم:

```
U[i] = i;
```



شکل ج ـ ۲ نمایش ساختمان‌داده مجموعه‌ی از هم‌جدا با درخت معکوس.

نمایش درختی ۱۰ مجموعه از هم جدا و پیاده‌سازی آن‌ها با آرایه، در شکل ج ـ ۳(الف) آمده است. نمونه‌ای از ادغام در شکل ج ـ ۳(ب) آمده است. وقتی مجموعه‌های {4} و {10} ادغام می‌شوند، گره حاوی ۱۰ را به عنوان فرزند گره حاوی ۴ قرار می‌دهیم. این کار با انتساب U[10] = 4 انجام می‌شود. به‌طورکلی، وقتی دو مجموعه را ادغام می‌کنیم، ابتدا مشخص می‌کنیم که بزرگ‌ترین اندیس در ریشه کدام درخت وجود دارد. شکل ج ـ ۳(ج) نمایش درختی و پیاده‌سازی آرایه‌ای متناظر با آن را پس از چند ادغام نشان می‌دهد. در این نقطه، فقط سه مجموعه از هم جدا وجود دارند. پیاده‌سازی رویه‌ها در ادامه آمده است، برای سهولت نمادگذاری در این‌جا و الگوریتم کروسکال (بخش ۲ـ۱ـ۴)، مجموعه جانی U را به عنوان پارامترِ روال‌ها درنظرنمی‌گیریم.

◆ ساختمان‌داده مجموعه جدا از هم (۱)

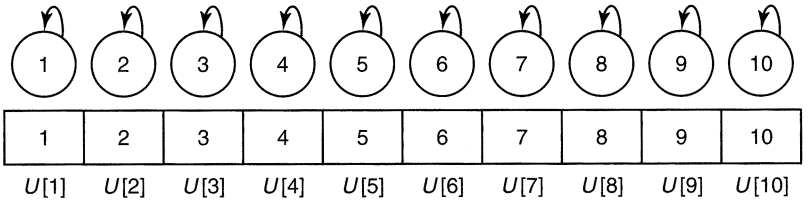
```
const int n = the number of elements in the universe;

typedef int index;
typedef index set_pointer;
typedef index universe[1..n]; // universe is indexed from 1 to n.

universe U;

void makeset (index i)
{
  U[i] = i;
}
```

```
set_pointer find (index i)
{
  index j;

  j = i;
  while (U[j]!=j)
     j = U[j];
  return j;
}

void merge (set_pointer p, set_pointer q)
{
  if (p < q)                          // p points to merged set;
     U[q] = p;                        // q no longer points to a set.
  else
     U[p] = q;                        // q points to merged set;
}                                     // p no longer points to a set.

bool equal (set_pointer p, set_pointer q)
{
  if (p == q)
     return true;
  else
     return false;
}

void initial (int n)
{
  index i;

  for (i = 1; i <= n; i++)
     makeset(i);
}
```

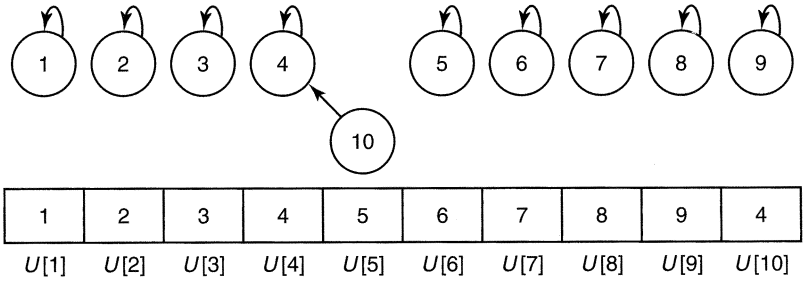
مقدار برگردانده‌شده توسط فراخوانی find(i)، اندیس ذخیره‌شده در ریشه‌ی درختِ حاوی i است. رویه initial می‌تواند n مجموعه جدا از هم را مشخص کند. زیرا چنین رویه‌ای در چنین مسئله‌ای ضروری است.

در بسیاری از الگوریتم‌هایی که از مجموعه‌های از هم جدا استفاده می‌کنند، n مجموعه از هم جدا را مقدار اولیه می‌دهیم و سپس m بار از حلقه عبور می‌کنیم (لازم نیست مقادیر m و n یکسان باشند). در داخل حلقه، رویه‌های equal، find و merge به تعداد ثابتی از دفعات فراخوانی می‌شوند. هنگام تحلیل الگوریتم، به پیچیدگی زمانی مقداردهی اولیه و حلقه تکرار برحسب n و m نیاز داریم. بدیهی است که پیچیدگی زمانی رویه initial در $\theta(n)$ است.

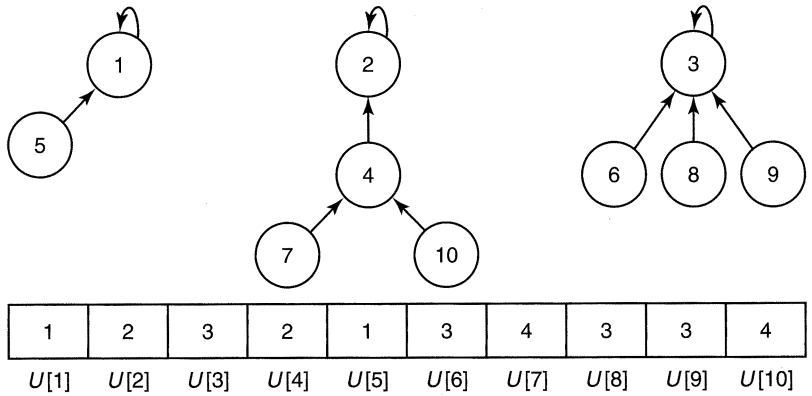
چون، مرتبه در اثر ضرب‌شدن در مقدار ثابت تغییر نمی‌کند، فرض می‌کنیم رویه‌های equal، find و merge در هر m گذر از حلقه، یک بار فراخوانی می‌شوند. بدیهی است که equal و merge در زمان ثابتی اجرا می‌شوند. فقط تابع find حاوی حلقه است. لذا، مرتبه پیچیدگی زمانی تمام فراخوانی‌ها تحت تأثیر تابع find قرار می‌گیرند. تعداد دفعات مقایسه در find را در بدترین حالت محاسبه می‌کنیم. به‌عنوان‌مثال، فرض کنید m = 5 است.



(الف) درخت‌های معکوس و پیاده‌سازی آرایه‌ای مربوط به ۱۰ مجموعه جدا از هم.



(ب) مجموعه‌های {4} و {0} در بخش (الف) ادغام شده‌اند. اکنون مقدار محل دهم آرایه ۴ است.



(ج) درخت‌های معکوس و پیاده‌سازی آرایه‌ای پس از چند ادغام. ترتیب ادغام‌ها، ساختار درخت‌ها را تعیین می‌کند.

شکل ج ـ ۳ پیاده‌سازی آرایه‌ای نمایش درخت‌های ساختمان‌داده مجموعه از هم جدا.

بدترین حالت وقتی رخ می‌دهد که دنباله‌هایی از ادغام زیر رخ دهد و پس از هر ادغام، تابع find را برای جست‌وجوی اندیس ۶ فراخوانی می‌کنیم:

$$
\begin{aligned}
&merge(\{5\},\ \{6\});\\
&merge(\{4\},\ \{5,6\});\\
&merge(\{3\},\ \{4,5,6\});\\
&merge(\{2\},\ \{3,4,5,6\});\\
&merge(\{1\},\ \{2,3,4,5,6\});
\end{aligned}
$$

(برای توضیح بیشتر، مجموعه‌های واقعی به صورت ورودی‌های تابع merge نوشته شده‌اند). درخت نهایی و پیاده‌سازی آرایه‌ای در شکل ج ـ ۴ آمده است. تعداد کل مقایسه‌ها در تابع find به صورت زیر است:

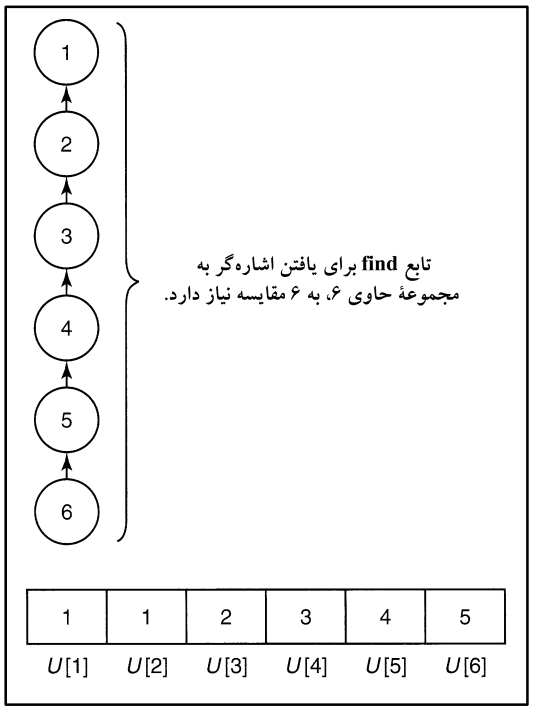
$$2+3+4+5+6$$

با تعمیم این نتیجه به یک مقدار دلخواه m، نتیجه می‌گیریم که تعداد مقایسه‌ها در بدترین حالت برابر است با:

$$2+3+\cdots+(m+1) \in \Theta\left(m^2\right)$$

تابع equal را درنظرنگرفتیم، زیرا آن تابع تأثیری در تعداد مقایسه‌ها در تابع find ندارد.

بدترین حالت وقتی رخ می‌دهد که مرتبه‌ای که در آن ادغام را انجام می‌دهیم، درختی را ایجاد کند که عمق آن یک واحد کمتر از تعداد گره‌های درخت باشد. اگر رویه merge را طوری اصلاح کنیم که این وضعیت رخ ندهد، می‌توانیم کارایی را بهبود ببخشیم. این کار را می‌توانیم با نگهداری عمق هر درخت و انتخاب درختی با عمق کمتر به عنوان ریشه، انجام دهیم. شکل ج ـ ۵ دو روش ادغام را با هم ادغام می‌کند. توجه کنید که روش جدید، درختی با عمق کمتر را ایجاد می‌کند. برای پیاده‌سازی این روش، لازم است عمق هر درخت در ریشه درخت ذخیره شود. پیاده‌سازی زیر، این کار را انجام می‌دهد.



شکل ج ـ ۴ نمونه‌ای از بدترین حالت در ساختمان‌داده از هم جدا (۱) وقتی که m = 5 است.

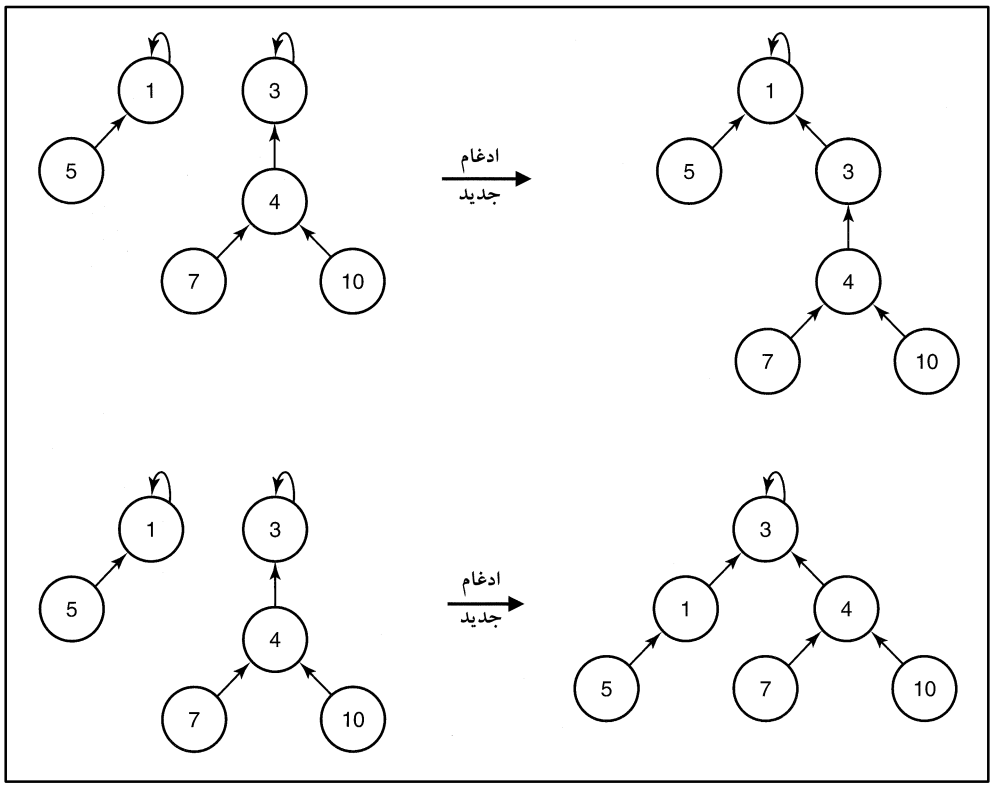
◆ ساختمان‌داده مجموعه جدا از هم (۲)

```
const int n = the number of elements in the universe;

typedef int index;
typedef index set_pointer;
struct nodetype
{
  index parent;
  int depth;
}
typedef nodetype universe[1..n];      // universe in indexed
                                      // from 1 to n.
universe U;
```

```
void makeset (index i);
{
  U[i].parent = i;
  U[i].depth = 0;
}

set_pointer find (index i)
{
  index j;

  j = i;
  while (U[j].parent != j)
     j = U[j].parent;
  return j;
}

void merge (set_pointer p, set_pointer q)
{
  if (U[p].depth == U[q].depth){
     U[p].depth = U[p].depth + 1;    // Tree's depth
     U[q].parent = p;                // must increase.
  }
  else if (U[p].depth < U[q].depth) // Make tree with lesser
     U[p].parent = q;                // depth the child.
  else
     U[q].parent = p;
}

bool equal (set_pointer p, set_pointer q)
{
  if (p == q)
     return true;
  else
     return false;
}

void initial (int n)
{
  index i;

  for (i = 1; i <= n; i++)
     makeset(i);
}
```

می‌توان نشان داد که تعداد مقایسه‌ها در بدترین حالت که در m گذر از حلقه‌ی حاوی تعداد ثابتی از فراخوانی‌های رویه‌های equal، find و merge است، برابر است با:

$$\theta(m \lg m)$$



شکل ج ـ ۵ در روش جدید ادغام، ریشه‌ای از درخت با عمق کوچک‌تر را به‌عنوان فرزند ریشه درخت دیگر درنظر می‌گیریم.

در بعضی از کاربردها، لازم است کوچک‌ترین عضو مجموعه با کارایی مناسبی پیدا شود. با استفاده از پیاده‌سازی اول، این کار آسان است، زیرا کوچک‌ترین عضو همیشه در ریشه درخت وجود دارد. به‌راحتی می‌توانیم آن پیاده‌سازی را اصلاح کنیم تا کوچک‌ترین عضو مجموعه را با کارایی مناسبی برگرداند. برای این کار، در ریشه هر درخت متغیری به‌نام smallest را قرار می‌دهیم. این متغیر حاوی کوچک‌ترین اندیس در درخت است. پیاده‌سازی زیر این کار را انجام می‌دهد.

◆ ساختمان‌داده مجموعه جدا از هم (۳)

```
const int n = the number of elements in the universe;

typedef int index;
typedef index set_pointer;

struct nodetype
{
  index parent;
  int depth;
  int smallest;
};
```

```
typedef nodetype universe[1..n];          // universe is indexed
                                          // from 1 to n.
universe U;

void makeset (index i)
{
  U[i].parent = i;
  U[i].depth = 0;
  U[i].smallest = i;                      // The only index i is smallest.
}

void merge (set_pointer p, set_pointer q)
{
  if (U[p].depth == U[q].depth){
     U[p].depth = U[p].depth + 1;         // Tree's depth must increase.
     U[q].parent = p;
     if (U[q].smallest < U[p].smallest)   // q's tree contains smallest
        U[p].smallest = U[q].smallest;    // index.
  }
  else if (U[p].depth < U[q].depth){      // Make tree with lesser depth
     U[p].parent = q;                     // the child.
     if (U[p].smallest < U[q].smallest)   // p's tree contains
         U[q].smallest = U[p].smallest;   // smallest index.
  }
  else{
     U[q].parent = p;
     if (U[q].smallest < U[p].smallest)   // q's tree contains
        U[p].smallest = U[q].smallest;    // smallest index.
  }
}

int small (set_pointer p)
{
  return U[p].smallest;
}
```

در این پیاده‌سازی، فقط رویه‌هایی را ارائه کردیم که با پیاده‌سازی دوم متفاوت بودند. تابع small کوچک‌ترین عضو مجموعه را برمی‌گرداند. چون زمان اجرای تابع small ثابت است، تعداد مقایسه‌های بدترین حالت در m گذر از حلقه‌ی حاوی تعداد ثابتی از فراخوانی‌های equal، find، merge و small، همانند الگوریتم نسخه (۲) است. یعنی در $\theta(m \lg m)$ است.

با استفاده از تکنیکی به‌نام **فشرده‌سازی مسیر**، می‌توان پیاده‌سازی‌ای را ایجاد کرد که تعداد مقایسه‌های بدترین حالت در m گذر از حلقه، تقریباً برحسب m خطی است. این پیاده‌سازی در براسارد و براتلی (۱۹۸۸) آمده است.